

Arhitectura Sistemelor de Calcul

Laborator - Partea 0x02

Ruxandra Bălucea
Bogdan Macovei
Cristian Rusu

Noiembrie 2024

Cuprins

1	Introducere	3
1.1	Context	3
1.2	Formatul instrucțiunilor și encodarea	4
1.3	Extensii RISC-V	5
1.4	Comparație x86 vs RISC-V	6
2	Mediul de lucru	7
3	Limbaajul Assembly RISC-V	9
3.1	Registrii arhitecturii	9
3.2	Tipuri de date	10
3.3	Instrucțiunile principale	10
3.4	Arhitectură load-store	11
3.5	Relocări	12
3.6	Apeluri de sistem	14
3.7	Salturi	14
3.8	Tablouri unidimensionale	16
3.9	RV32C - instrucțiuni comprimate	16
4	Convenția de apel	18
4.1	Apelul unei proceduri	18
4.2	Argumentele unei proceduri și valoarea de retur	19
4.3	Crearea cadrului de apel	20
5	Pipelining	23
5.1	Hazard-uri în pipeline	24
5.2	Măsurarea performanței	26
6	Cache	27
6.1	Performanța memoriei cache	30

7	Exercitii	31
7.1	Laboratorul 9	31
7.2	Laboratorul 10	31
7.3	Laboratorul 11	31
8	Resurse suplimentare	32

1 Introducere

1.1 Context

RISC-V este un set de instrucțiuni (**ISA** - *Instructions Set Architecture*) RISC dezvoltat în 2010 la Universitatea Berkeley din California ce prezintă următoarele caracteristici principale:

1. Este o arhitectură **open-source**. Spre deosebire de x86 și ARM, oricine poate utiliza și implementa RISC-V fără taxe de licențiere sau restricții, promovând inovarea și reducând dependența de furnizorii comerciali.
2. **Modularitate**. RISC-V definește un set de instrucțiuni de bază minimalist la care se pot adăuga extensii opționale (pentru operații în virgulă mobilă, vectori, criptografie, cache management etc).
3. **Portabilitate**. Arhitectura este neutră din punct de vedere hardware, ceea ce o face ușor de implementat pe diverse tipuri de dispozitive. RISC-V poate fi utilizat atât în procesoare simple de IoT, cât și în procesoare complexe pentru servere și supercomputere.

Comunitatea globală de dezvoltatori contribuie activ la standardizarea și implementarea RISC-V, iar organizații precum **RISC-V International**¹ coordonează aceste eforturi. Dintre membrii fondatori amintim: Andes, Antmicro, ETH Zurich, Google, IBM, lowRISC, Microchip, Qualcomm, Rambus, Rumble, SiFive etc.

Inovația accelerată, independența tehnologică și costurile reduse fac din RISC-V alegerea a numeroase companii pentru dispozitive embedded, mobile, centre de date și supercomputere, inteligență artificială, automotive și chiar aplicații militare și guvernamentale.

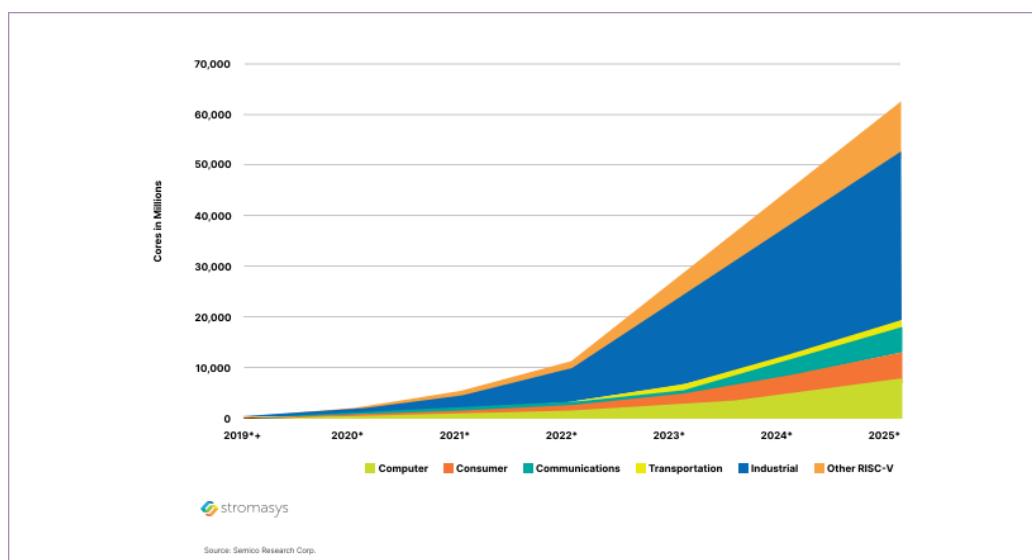


Figura 1: Evolutia RISC-V ²

¹<https://riscv.org/>

²<https://www.stromasys.com/resources/all-about-the-risc-v-processors/>

1.2 Formatul instrucțiunilor și encodarea

Arhitecturile RISC-V și x86 adoptă abordări fundamental diferite în ceea ce privește formarea instrucțiunilor și codificarea lor, reflectând filosofia generală a designului lor. În timp ce RISC-V este simplă și modulară, cu un set fix de formate de instrucțiuni, x86 este o arhitectură mai complexă și mai veche, cu codificări variabile.

Codificarea instrucțiunilor x86 ocupă între 1 și 15 octeți și poate include diferite componente ce fac decodarea mai lentă și mai complicată. Dintre acestea amintim:

- **prefixuri** (de exemplu pentru diferențierea dimensiunii registrului - **mov %eax, %ebx** are encodarea **89 c3**, pe cand **mov %ax, %bx**) are encodarea **66 89 c3**;
- **opcode-uri** (mnemonica) - specifică instrucțiunea principală;
- **moduri de adresare** - registru/memorie;
- **operandi** suplimentari.

Pe de cealaltă parte, instrucțiunile RISC-V au lungime fixă de 32 de biți (4 bytes) și respectă una dintre cele câteva formate standard (de exemplu, R, I, S, B, U, J), ceea ce face decodarea rapidă și simplă pentru hardware.

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12:10:5]				rs2		rs1		funct3		imm[4:1:11]		opcode		B-type
imm[31:12]										rd		opcode		U-type
imm[20:10:1:11:19:12]										rd		opcode		J-type

Figura 2: Formatul instrucțiunilor RISC-V ³

Exemplu: Se consideră instrucțiunea **add a0, a1, a2** care realizează suma dintre valorile regăsite în regiștrii **a1**, respectiv **a2** și o depune în **a0**. Aceasta este o instrucțiune de tip R, având definite în manualul cu specificațiile arhitecturii⁴ **func7 = 0000000**, **func3 = 000** și **opcode = 0110011**. În plus știm codurile regiștrilor ca fiind următoarele: (în decimal) **a0 = 10**, **a1 = 11**, **a2 = 12**. Luând în considerare informațiile anterioare, encodarea instrucțiunii va fi următoarea:

func7	rs2	rs1	funct3	rd	opcode
0 0 0 0 0 0 0	0 1 1 0 0	0 1 0 1 1	0 0 0	0 1 0 1 0	0 1 1 0 0 1 1

Așadar, grupând câte 4 biți și convertind în hexa, o să obținem encodarea **0x00c58533**.

Exercițiu: Care este encodarea pentru instrucțiunea **addi a0, a1, 15** care este de tip I și realizează suma dintre imediatul **15** și valoarea din registrul sursă **a1**, depozitând valoarea în registrul destinație **a0** (vezi codurile regiștrilor mai sus)? De asemenea, se cunosc **func7 = 0000000**, **func3 = 000** și **opcode = 0010011**.

³<https://five-embeddev.com/riscv-user-isa-manual/Priv-v1.12/instr-table.html>

⁴<https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>

1.3 Extensii RISC-V

Așa cum am prezentat anterior, una dintre principalele avantaje ale arhitecturii este modularitatea și împărțirea instrucțiunilor pe funcții specifice care să poată fi folosite sau nu în acord cu scopul procesorului.

La nucleul RISC-V se află setul de instrucțiuni de bază, cunoscut sub denumirea de **RV32I** pentru arhitecturi pe 32 de biți și **RV64I** pentru arhitecturi pe 64 de biți. Acestea includ un set minim de instrucțiuni necesar pentru a implementa o arhitectură de procesor funcțională. Caracteristicile principale ale bazei includ:

- Operații aritmetice de bază (add, sub etc);
- Operații de acces la memorie (lw, sw etc.);
- Instrucțiuni de control-flow (jal, beq etc)

De asemenea, dintre extensiile standardizate amintim:

- **RV32M** - suport pentru operații aritmetice mai avansate, cum ar fi multiplicarea (mul) și împărțirea (div). Este esențială pentru aplicații care necesită calcul numeric intensiv.
- **RV32C** - subset de instrucțiuni compresate, reducând dimensiunea codului și economisind memorie. Este utilă în sisteme integrate sau alte aplicații cu resurse limitate.
- **RV32A** - suport pentru operații atomice, care sunt cruciale în programele multithread. Instrucțiunile precum lr (load-reserved) și sc (store-conditional) permit sincronizarea eficientă între mai multe fire de execuție.
- **RV32F** și **RV32D** - suport pentru operații în virgulă mobilă. Acestea sunt esențiale în aplicații precum procesarea semnalelor digitale și învățarea automată.
- **RV32G** - este o extensie generală ce este echivalentă cu RV32IMAFD.
- **RV32V** - suport pentru calcul vectorial, esențial în aplicații precum procesarea imaginilor, simulările științifice și deep-learning.
- **RV32B** - instrucțiuni optimizate pentru manipularea biților, folosite în criptografie și procesarea semnalelor

La fel ca setul de bază, toate aceste extensii suportă și variante pe 64 de biți.

În plus, organizațiile și companiile pot defini propriile extensii pentru nevoi specifice, contribuind la diversitatea implementărilor RISC-V.

Toate extensiile ratificate, precum și statusul curent al extensiilor în curs de ratificare pot fi consultate aici⁵.

⁵<https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154732/Ratified+Extensions>

1.4 Comparație x86 vs RISC-V

În finalul acestei secțiuni de introducere, prezentăm un tabel cu principalele avantaje și dezavantaje ale celor 2 arhitecturi studiate. Acest tabel poate fi privit chiar ca o comparație CISC vs RISC.

	x86 (CISC)	RISC-V (RISC)
Avantaje	<ul style="list-style-type: none">• Codificare compactă pentru instrucțiuni simple → economisirea memoriei în aplicații vechi.• Suport pentru moduri complexe de adresare, util pentru software legacy.	<ul style="list-style-type: none">• Simplitate și uniformitate → ușor de implementat în hardware și rapid de decodat.• Scalabilitate → extensiile pot fi adăugate fără a afecta codificările existente.• Pipeline eficient → formatele fixe permit instrucțiuni care pot fi decodate în paralel.
Dezavantaje	<ul style="list-style-type: none">• Decodare lentă și complicată → dificultăți în proiectarea procesoarelor moderne.• Moștenirea istorică → dificultăți în a elimina instrucțiunile redundante sau ineficiente.	<ul style="list-style-type: none">• Uneori necesită mai multe instrucțiuni pentru operațiuni complexe, comparativ cu x86.

Tabela 1: Comparație cu avantaje și dezavantaje între x86 și RISC-V

2 Mediul de lucru

Arhitectura RISC-V beneficiază de un ecosistem divers de tool-uri și simulatoare care acoperă nevoi educaționale, de cercetare și industriale. Mai jos sunt principalele tool-uri și simulatoare utilizate în ecosistemul RISC-V, împărțite în categorii relevante.

Simulatorul oficial este **Spike**⁶ și este dezvoltat de RISC-V International, având suport pentru toate extensiile standard și putând fi folosit pentru testare și validare. Acesta este destul de lent, principalul focus fiind pe acuratețe. Pe de altă parte, **QEMU**⁷ este un **emulator** și suportă mai multe arhitecturi (ARM, X86 etc.) și mai multe modalități de configurare de sistem, fiind potrivit pentru dezvoltarea de sisteme de operare și aplicații embedded. Permite integrarea de tool-uri precum **gdb** și este mult mai rapid, traducerea codului făcându-se dinamic din ISA-ul emulat în ISA-ul nativ al mașinii.

Un subdomeniu extrem de important pentru dezvoltarea ecosistemului RISC-V este reprezentat de design-ul hardware. Dintre cele mai cunoscute limbaje utilizate pentru proiectarea și validarea procesoarelor amintim **Chisel**⁸ care permite definirea de scheme hardware pentru design-ul de procesoare. De asemenea, ca o platformă ce înglobează multe dintre procesoarele astfel dezvoltate, **Chippyard**⁹ oferă suport pentru diverse core-uri (RocketChip, BOOM etc.) și acceleratoare, oferind de asemenea posibilitatea rulării în cloud pe un **FPGA** (circuit fizic reprogramabil pe baza unei scheme a unui sistem).

De asemenea, dintre cele mai importante tool-uri software dezvoltate amintim:

- **RISC-V GNU Toolchain**¹⁰ - compiler GCC, binutils, și GDB pentru RISC-V.
- **LLVM/Clang pentru RISC-V**¹¹ - compilare optimizată, suport extins pentru extensii ISA.

Majoritatea acestor ustensile de lucru sunt necesare a fi build-uite și adaptate pentru mediul de lucru al fiecăruia, proces ce poate dura și câteva ore. Din acest motiv, în cadrul laboratorului vom utiliza un simulator online cu o interfață user-friendly, simulator proiectat pentru utilizarea în scop educațional - **Ripes** - <https://ripes.me/>. Sursele pentru acest simulator care poate fi și instalat local se găsesc aici¹². De asemenea, un tutorial amănunțit despre funcționalitățile simulatorului pot fi regăsite în secțiunea de documentație.

Acesta permite vizualizarea execuției instrucțiunilor într-un procesor simplificat, fie single-cycle, fie într-un pipeline. Oferă o interfață intuitivă pentru explorarea etapelor procesorului, modificarea regiștrilor, memoriei și stivei și rularea programelor în assembly RISC-V.

Ripes implementează extensia de bază RV32I, permițând și activarea extensiilor C și M.

Vom reveni asupra altor detalii despre acest simulator într-o secțiune ulterioară, punctând acum doar ce este necesar pentru a observa conceptele studiate anterior:

⁶<https://github.com/riscv-software-src/riscv-isa-sim>

⁷<https://www.qemu.org/>

⁸<https://www.chisel-lang.org/docs>

⁹<https://chipyard.readthedocs.io/en/stable/Chipyard-Basics/Chipyard-Components.html>

¹⁰<https://github.com/riscv-collab/riscv-gnu-toolchain>

¹¹<https://github.com/llvm/llvm-project>

¹²<https://github.com/mortbopet/Ripes>

- **Secțiunile statice (.text, .data, .bss)** pot fi navigate prin butonul `Go to section:`.
- **Stiva** poate fi vizualizată prin folosirea butonului `Go to register:` și alegerea registrului `sp` care menține (similar cu `esp`), vârful stivei. Remarcăm poziționarea acesteia la o adresă mare `0x7ffff0`, așa cum am discutat spațiul de adresă al unui proces în cadrul laboratorului despre proceduri pentru arhitectura x86.
- **Registrii** - aceștia pot fi vizualizați în tabelul din dreapta paginii.
- **Codul** - scrierea se face direct în fereastră putând observa o conversie instantă în binar, precum și varianta dezasamblată (encodată și ulterior decodată prin aplicarea succesivă a unui asamblor și a unui dezasamblor). De asemenea, se pot și încărca direct fișiere scrise local.

Atenție!

1. Pentru rularea instrucțiunilor pas cu pas, este nevoie să schimbați procesorul din iconița de sub *File*, într-un **Single-cycle processor**.
2. Uneori site-ul refuza să se încarce. Aceasta problema poate fi rezolvată prin curățarea memoriei cache sau prin deschiderea simulatorului într-o fereastră incognito.

3 Limbajul Assembly RISC-V

3.1 Regiștrii arhitecturii

RISC-V dispune de 32 de regiștrii generali (x0 până la x31), fiecare având o dimensiune de 32 de biți (pentru arhitectura de 32 de biți). Acești regiștri sunt utilizați pentru manipularea datelor, fiind folosiți la fel ca la x86 ca niște variabile.

Encodare	Encodare compri-mată	Registru	Nume ABI	Descriere	Callee-saved
0	-	x0	zero	Hardwired zero	-
1	-	x1	ra	Return address	Da
2	-	x2	sp	Stack pointer	Da
3	-	x3	gp	Global pointer	-
4	-	x4	tp	Thread pointer	-
5	-	x5	t0	Temporary register 0	Nu
6	-	x6	t1	Temporary register 1	Nu
7	-	x7	t2	Temporary register 2	Nu
8	0	x8	s0/fp	Saved register 0 / frame pointer	Da
9	1	x9	s1	Saved register 1	Da
10	2	x10	a0	Function argument 0 / return value 0	Nu
11	3	x11	a1	Function argument 1 / return value 1	Nu
12	4	x12	a2	Function argument 2	Nu
13	5	x13	a3	Function argument 3	Nu
14	6	x14	a4	Function argument 4	Nu
15	7	x15	a5	Function argument 5	Nu
16	-	x16	a6	Function argument 6	Nu
17	-	x17	a7	Function argument 7	Nu
18	-	x18	s2	Saved register 2	Da
19	-	x19	s3	Saved register 3	Da
20	-	x20	s4	Saved register 4	Da
21	-	x21	s5	Saved register 5	Da
22	-	x22	s6	Saved register 6	Da
23	-	x23	s7	Saved register	Da
24	-	x24	s8	Saved register 8	Da
25	-	x25	s9	Saved register 9	Da
26	-	x26	s10	Saved register 10	Da
27	-	x27	s11	Saved register 11	Da
28	-	x28	t3	Temporary register 3	Nu
29	-	x29	t4	Temporary register 4	Nu
30	-	x30	t5	Temporary register 5	Nu
31	-	x31	t6	Temporary register 6	Nu

Tabela 2: Regiștrii de uz general RISC-V

Un alt registru de interes este registrul **pc (program counter)** care menține întotdeauna adresa instrucțiunii ce urmează a fi executată.

3.2 Tipuri de date

Tipurile principale de date pe care le vom utiliza sunt

1. **byte** - după cum îi spune și numele, ocupă 1 byte, adică 8 biți în memorie;
2. **halfword** - ocupă 2 bytes (16 de biți) și este utilizat pentru stocarea numerelor fracționare;
3. **word** - ocupă 4 bytes (32 biți) și este utilizat pentru stocarea întregilor. **long** ocupă tot 32 de biți.
4. **doubleword** - ocupa 8 bytes (64 biți);
5. **ascii**, **asciz** și **space** cu aceeași semnificație ca pe x86 (**space** nu este implementat de Ripes).

3.3 Instrucțiunile principale

Instrucțiune	Operanzi	Descriere
add	rd, rs1, rs2	Adună $rs1 + rs2$ și salvează în rd.
sub	rd, rs1, rs2	Scade rs2 din rs1 și salvează în rd.
mul	rd, rs1, rs2	Înmulțește rs1 cu rs2 (M-extension).
div	rd, rs1, rs2	Împarte rs1 la rs2 (M-extension).
rem	rd, rs1, rs2	Restul împărțirii întregi între rs1 și rs2.
and	rd, rs1, rs2	Realizează and logic între rs1 și rs2.
or	rd, rs1, rs2	Realizează or logic între rs1 și rs2.
xor	rd, rs1, rs2	Realizează xor logic între rs1 și rs2.
sll	rd, rs1, rs2	Shift la stânga logic cu valoarea din rs2.
srl	rd, rs1, rs2	Shift la dreapta logic cu valoarea din rs2.
addi	rd, rs1, imm	Adună $rs1 + imm$ și salvează în rd.
slti	rd, rs1, imm	Compară $rs1 < imm$ și setează rd la 1 sau 0.
andi	rd, rs1, imm	Realizează and între rs1 și imm.
lui	rd, imm	Încarcă imm pe cei 20 de biți superiori din rd ($rd = imm \ll 12$)
auipc	rd, imm	Încarcă imm pe cei 20 de biți superiori din rd, adunând la această valoare și pc ($rd = pc + (imm \ll 12)$).

Tabela 3: Principalele instrucțiuni RISC-V și operanzii lor

De asemenea, există și câteva pseudoinstrucțiuni importante. O pseudoinstrucțiune este o instrucțiune care poate fi folosită în asamblare, dar care nu are o encodare proprie, ci este de fapt convertită în una sau mai multe instrucțiuni clasice.

Pseudoinstrucție	Instrucțiuni echivalente	Descriere
<code>li rd, imm</code>	<code>lui rd, imm[31:12]</code> <code>addi rd, rd, imm[11:0]</code>	Încarcă valoarea imediată <code>imm</code> în registrul <code>rd</code> .
<code>la rd, label</code>	<code>lui rd, label[31:12]</code> <code>addi rd, rd, label[11:0]</code>	Încarcă adresa etichetei <code>label</code> în registrul <code>rd</code> .
<code>move rd, rs</code>	<code>addi rd, rs, 0</code>	Copiază valoarea din registrul <code>rs</code> în registrul <code>rd</code> .
<code>nop</code>	<code>addi x0, x0, 0</code>	Instrucțiune ”no operation”, nu face nimic.

Tabela 4: Principalele pseudoinstrucțiuni RISC-V

3.4 Arhitectură load-store

RISC-V este o arhitectură bazată pe principiul load-store, ceea ce înseamnă că toate operațiile de procesare a datelor trebuie să utilizeze exclusiv registre pentru calcule, iar memoria poate fi accesată doar prin instrucțiuni dedicate de încărcare (load) și stocare (store). Această abordare este esențială în designul arhitecturilor RISC (Reduced Instruction Set Computing).

Deoarece toate operațiile implică doar regiștri, numărul de etape în pipeline-ul procesorului este redus, ceea ce duce la o execuție mai eficientă a instrucțiunilor. Memoria este accesată doar în etapele de load și store, reducând complexitatea logicii de execuție.

Pentru încărcarea unei valori din memorie vom folosi instrucțiunea:

```
lw reg1, offset(reg2)
```

unde `reg2` reține o adresă de memorie, `offset` reprezintă numărul de bytes adăugați pentru a ajunge pe poziția din care vrem să citim, iar `reg1` este registrul în care se va încărca valoarea din memorie. Pentru stocarea unei valori în memorie vom folosi instrucțiunea:

```
sw reg1, offset(reg2)
```

unde `offset` și `reg2` sunt folosiți similar pentru a calcula adresa din memorie la care se face stocarea, iar `reg1` este registrul din care se ia valoarea ce urmează a fi depozitată.

Aceste variante de instrucțiuni se aplică pe o dimensiune de 32 de biți. Similar există instrucțiuni pentru a lucra cu 16 biți (`lh` și `sh`), respectiv cu 8 biți (`lb` și `sb`).

Registrul `gp` este folosit pentru a reține adresa secțiunii `.data`. Una dintre modalitățile de accesare a datelor din memorie, este așadar prin folosirea unui offset raportat la baza secțiunii.

Exercițiu: Pentru o exemplificare, să considerăm secvența următoare de cod. Cum va arăta memoria în urma execuției celor 2 instrucțiuni? Ce observații legat de declarațiile din zona `.data`?

```
.data
    .long 5
    .long 6

.text
.global main
main:
    lw a0, 0(gp)
    sw a0, 4(gp)
```

3.5 Relocări

Pentru a putea folosi în continuare simbolii declarați în memorie în cadrul instrucțiunilor și a nu fi nevoiți să calculăm de fiecare offset-ul pentru accesarea datelor, este nevoie să ne uităm peste conceptul de relocare.

O relocare este un proces utilizat în timpul compilării, linkării sau încărcării unui program pentru a ajusta adresele de memorie sau referințele la simboluri, astfel încât programul să poată fi executat corect indiferent de poziția sa în memorie. Majoritatea operanzilor de tip imediat acceptă o variantă de relocare pentru deducerea ulterioară a respectivului offset. Dacă poziționarea secțiunilor este statică și simbolii sunt prezenți în sursa curentă, adresa lor se poate deduce în timpul procesului de compilare, adresa fiind calculată automat. Dacă simbolii în schimb sunt necunoscuți și sunt aflați în timpul procesului de linkare (sunt declarați în alte surse, biblioteci etc), iar pozițiile secțiunilor sunt fixe (în special pentru dispozitivele embedded unde sunt predefinite printr-un script de configurare), adresa este calculată în această etapă. Altfel, în cazul în care secțiunile sunt dinamice (poziționate la diferite adrese la fiecare rulare), relocările vor fi rezolvate abia la rulare.

Pentru utilizarea simboluri din zonele de date, RISC-V folosește următoarele relocări:

- **Relocări pentru partea superioară (`%hi` și `%pcrel_hi`).** Cele două sunt folosite în cadrul instrucțiunilor lui, respectiv `auipc` pentru a prelua cei 20 de biți superior dintr-o adresă.
- **Relocări pentru offset-uri (`%lo` și `%pcrel_lo`).** Cele două sunt folosite pentru instrucțiunile `lw` și `sw` pentru adăugarea offset-ului de 12 biți rămași. De asemenea, poate fi folosită și în cadrul unei instrucțiuni `addi` pentru a ajuta la calcularea adresei de început (pentru un vector, un string). Această relocare `lo` este precedată întotdeauna de o instrucțiune cu relocare `hi`, obținând prin această combinație adresa pe 32 de biți.
- **Relocări pentru salturi** - vor fi prezentate în secțiunea despre salturi.
- Altele.

De asemenea, o diferențiere importantă o reprezintă modul în care funcționează relocările absolute, respectiv relocările relative la program counter:

- **`%lo(sym)`**, respectiv **`%hi(sym)`** vizează exact biții inferiori, respectiv biții superiori ai adresei simbolului `sym`;

- `%pcrel_hi(sym)` face referire $(\text{adr_sym} - \text{pc}) \gg 12$ (biții superiori);
- `%pcrel_lo(label)` unde `label` indică spre o instrucțiune având o relocare de tip `%pcrel_hi(sym)` va considera $(\text{adr_sym} - \text{label}) \% 2^{12}$ (biții inferiori).

Astfel, pentru a obține adresa unei variabile, putem folosi una dintre variantele de mai jos:

```
.data
a: .long 5

.text
.global main
main:
    # Varianta 1 (statică)
    lui a0, %hi(a)
    lw a1, %lo(a)(a0)

    # Varianta 2 (relativă cu relocări)
    l0:
    auipc a2, %pcrel_hi(a)
    lw a3, %pcrel_lo(l0)(a2)

    # Varianta 3 (convenție de scriere care se expandează la varianta 2)
    lw a4, a

    # Varianta 4 (relativă cu global pointer)
    lw a5, 0(gp)
```

Observație! Programul funcționează și fără un apel `exit(0)` în cazul acestui simulator, neexistând un sistem de operare.

Întrucât pentru Ripes cunoaștem de la început locația secțiunilor `.data` și `.text` și de asemenea simbolul `a` este în același fișier, putem aplica toate variantele de mai sus. Vom alege în continuare să folosim varianta 3 pentru programele pe care le vom avea de implementat. **Relocările relative sunt într-o variantă instabilă pe Ripes și nu vor funcționa corect!**

Exercițiu: Analizați cele 4 variante, observând ce adrese sunt reținute la fiecare pas. Care este diferența dintre relocările `%hi` și `%pcrel_hi`, respectiv `%lo` și `%pcrel_lo`? De ce variantele relative pot fi utilizate și când nu se cunosc static adresele secțiunilor? Observați comportamentul greșit al variantei 2 (comparativ cu varianta 3).

Exercițiu: Transformați variantele 1, 2 și 4 de mai sus astfel încât în regiștrii `a1`, `a3` și `a5` să obțineți adresa lui `a` (în loc de valoarea lui) folosind instrucțiunea `addi`.

Exercițiu: Transformați variantele de mai sus pentru stocare și verificați ce variantă este corect implementată de Ripes. **Observație!** Toate variantele de mai sus sunt corecte arhitectural vorbind, stadiul dezvoltării tool-ului fiind singurul impediment în folosirea lor.

3.6 Apeluri de sistem

Pentru folosirea apelurilor de sistem mecanismul este similar cu ce am văzut pe x86, diferind doar regiștrii. Astfel, pentru a apela o funcție sistem vom folosi instrucțiunea

```
ecall
```

care va produce verificarea regiștrilor următori:

- **a7** - pentru stocarea codului funcției
- **a0** - **a6** - pentru încărcarea argumentelor.

De exemplu, funcția **exit** are codul 93. Deci pentru realizarea apelui **exit(0)**, vom avea secvența următoare de mai jos.

```
li a7, 93
li a0, 0
ecall
```

Codurile de apel pentru **Ripes** pot fi găsite în secțiunea **Help > System calls**.

Exercițiu: Declarați un string și un întreg în memorie și afișați-le pe ecran.

3.7 Salturi

O instrucțiune importantă pentru efectuarea de salturi necondiționate este instrucțiunea:

```
jal rd, imm
```

care stochează valoarea **pc + 4** în registrul **rd** și face saltul la adresa **pc + imm**.

În general pentru executarea de salturi necondiționate vom folosi:

```
j label
```

care este de fapt o pseudoinstrucțiune pentru instrucțiunea:

```
jal x0, label
```

Cum registrul **x0** nu poate fi modificat, asta înseamnă că valoarea **pc + 4** nu se salvează nicăieri, iar saltul se face la adresa lui **label**.

Fiind vorba din nou despre lucrul cu simbolii, avem și în acest caz o relocare specifică instrucțiunii - **R_RISCV_JAL** care indică înlocuirea simbolului cu adresa sa relativă la **pc**.

Pentru salturile condiționate, spre deosebire de x86, RISC-V comasează instrucțiunea de comparare și de salt într-una singură.

În mod similar, putem folosi în loc de imediat un simbol a cărui adresă va fi calculată relativ prin rezolvarea relocării **R_RISCV_BRANCH**.

Să considerăm următorul exemplu:

Instrucțiune	Operanzi	Descriere
blt(u)	rs1, rs2, imm	Dacă $rs1 < rs2$, salt la $pc + imm$
ble(u)	rs1, rs2, imm	Dacă $rs1 \leq rs2$, salt la $pc + imm$
bgt(u)	rs1, rs2, imm	Dacă $rs1 > rs2$, salt la $pc + imm$
bge(u)	rs1, rs2, imm	Dacă $rs1 \geq rs2$, salt la $pc + imm$
beq	rs1, rs2, imm	Dacă $rs1 == rs2$, salt la $pc + imm$
bne	rs1, rs2, imm	Dacă $rs1 \neq rs2$, salt la $pc + imm$

Tabela 5: Principalele instrucțiuni RISC-V pentru salturi condiționate.

Pseudoinstrucțiune	Operanzi	Descriere
beqz	rs1, imm	Dacă $rs1 == 0$, salt la $pc + imm$
bnez	rs1, imm	Dacă $rs1 \neq 0$, salt la $pc + imm$

Tabela 6: Principalele pseudoinstrucțiuni RISC-V pentru salturi condiționate.

```

.data
.long 6
.long 7
egale: .asciz "Numerele sunt egale\n"
diferite: .asciz "Numerele sunt diferite\n"
.text
.global main
main:

    lw a0, 0(gp)
    lw a1, 4(gp)

    beq a0, a1, 10
    li a7, 4
    la a0, diferite
    ecall
    j final

10:
    li a7, 4
    la a0, egale
    ecall

final:
    li a7, 93
    li a0, 0
    ecall

```

Exercițiu: Analizați programul și conversiile simbolilor în imediați.

3.8 Tablouri unidimensionale

Mecanismul de funcționare este același ca pe x86, dar nu există o grupare specifică pentru accesarea elementelor.

Pentru clarificare, să considerăm următorul exemplu care realizează suma elementelor dintr-un vector de întregi și o depozitează în memorie:

```
.data
n: .long 5
s: .long 0
v: .long 1, 2, 3, 4, 5

.text
.global main
main:
    lw a0, 0(gp)      # se incarca n in a0
    addi a1, gp, 8     # se pune adresa lui v in a1
    li a2, 0           # in a2 vom calcula suma
    li a3, 0           # in a3 vom mentine indexul

    begin:
    bge a3, a0, final

    slli a4, a3, 2     # a4 = 4 * index
    add a4, a4, a1     # a4 = adresa lui v + 4 * index
    lw a4, 0(a4)       # a4 = *(adresa lui v + 4 * index)
    add a2, a2, a4     # adunam elementul la suma totala
    addi a3, a3, 1     # incrementam indexul
    j begin

    final:
    sw a2, 4(gp)

    li a7, 93
    li a0, 0
    ecall
```

Exercițiu: Să se rescrie exemplul anterior, modificând întâi valoarea lui `n` în `n * 4` și incrementând apoi indexul cu 4 la fiecare iterație.

3.9 RV32C - instrucțiuni comprimate

Pentru a îmbunătăți performanța și eficiența memoriei, RISC-V include și un subset de instrucțiuni comprimate, care permit utilizarea a 16 biți pentru a reprezenta anumite instrucțiuni, reducând astfel dimensiunea programelor și economisind lățimea de bandă a memoriei.

RVC realizează o schemă simplă de conversie care permite scurtarea instrucțiunilor din setul de bază când:

1. imediatul/offsetul este suficient de mic;
2. unul dintre regiștri este x0 (zero register), x1 (return address) sau x2 (stack pointer-ul);
3. este folosit același registru pentru destinație și pentru sursă;
4. se folosesc regiștrii care au o variantă comprimabilă (vezi Tabelul 2)

Exemple

```
lw t0, 4(sp)    --> c.lwsp t0, 4(sp)    // este folosit registrul sp
addi t0, t0, 1  --> c.addi t0, 1        // sursa == destinație == t0
lw a0, 4(s0)    --> c.lw a0, 4(s0)     // a0 și s0 au variante comprimabile
lw t0, 516(sp)  --> nu se comprima     // imediatul este prea mare
```

Astfel, sunt definite următoarele formate pentru encodarea instrucțiunilor comprimate:

Format	Meaning	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CR	Register	funct4				rd/rs1				rs2				op			
CI	Immediate	funct3		imm		rd/rs1				imm				op			
CSS	Stack-relative Store	funct3		imm				rs2				op					
CIW		funct3		imm								rd'		op			
CL	Load	funct3		imm		rs1'		imm		rd'		op					
CS	Store	funct3		imm		rs1'		imm		rs2'		op					
CB	Branch	funct3		offset		rs1'		offset				op					
CJ	Jump	funct3		jump target												op	

Figura 3: Formatul instrucțiunilor RVC ¹³

Opcode-urile acestor operații au fost gândite astfel încât să nu se suprapună cu cele ale instrucțiunilor uzuale. Sunt verificați parte dintre biții de identificare, iar procesorul trimite instrucțiunea către un decodor specializat care interpretează această variantă pe 16 biți. În rest, pipelineul rămâne neschimbat.

Principalele instrucțiuni sunt indicate mai jos. De asemenea, un tabel cu toate instrucțiunile comprimate poate fi găsit aici ¹⁴

Pentru activarea acestor instrucțiuni în Ripes este nevoie să bifați și extensia C în meniul de unde ați ales tipul de procesor.

Exercițiu: Să se rescrie exemplul din Subsecțiunea 3.8, parcurgând șirul de la final spre început și folosind ca index inițial $4 * (n - 1)$. Ce dimensiune are zona `.text`?

Exercițiu: Să se rescrie codul anterior folosind instrucțiuni comprimate. Care este noua dimensiune a zonei de cod?

¹³<https://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-209.pdf>

¹⁴<https://five-embeddev.com/riscv-user-isa-manual/Priv-v1.12/rvc-instr-table.html>

Instrucțiune	Operanzi	Descriere
<code>c.addi</code>	<code>rd, imm</code>	Adaugă un imediat la registrul <code>rd</code> .
<code>c.addis</code>	<code>rd, imm</code>	Adaugă un imediat (16 biți) la registrul <code>sp</code> (pointerul de stivă).
<code>c.add</code>	<code>rd, rs</code>	Adaugă valoarea din registrul <code>rs</code> la registrul <code>rd</code> .
<code>c.lwsp</code>	<code>rd, offset(sp)</code>	Încarcă un cuvânt din memorie la adresa <code>sp + offset</code> în registrul <code>rd</code> .
<code>c.swsp</code>	<code>rd, offset(sp)</code>	Stochează un cuvânt din registrul <code>rd</code> la adresa <code>sp + offset</code> .
<code>c.li</code>	<code>rd, imm</code>	Încarcă un imediat de 16 biți în registrul <code>rd</code> .
<code>c.j</code>	<code>offset</code>	Salt necondiționat la adresa specificată de <code>offset</code> .
<code>c.beqz</code>	<code>rs, offset</code>	Salt condiționat dacă registrul <code>rs</code> este zero.
<code>c.bnez</code>	<code>rs, offset</code>	Salt condiționat dacă registrul <code>rs</code> nu este zero.
<code>c.not</code>	<code>rd, rs</code>	Neagă (complementul pe biți) valoarea din registrul <code>rs</code> și o stochează în registrul <code>rd</code> .
<code>c.lw</code>	<code>rd', offset(rs')</code>	Încarcă un cuvânt din memorie la adresa <code>rs' + offset</code> în registrul <code>rd'</code> .
<code>c.sw</code>	<code>rd', offset(rs')</code>	Stochează un cuvânt din registrul <code>rd'</code> la adresa <code>rs' + offset</code> .

Tabela 7: Principalele instrucțiuni RVC și operanzii lor

4 Convenția de apel

Pentru alocarea de spațiu pe stivă este necesar să folosim instrucțiuni aritmetice și de operare cu memoria. Deși nu există instrucțiuni precum `push` și `pop`, comportamentul este același.

De exemplu, pentru a stoca valoarea din registrul `s0` pe stivă, vom folosi următoarea secvență:

```
addi sp, sp, -4
sw s0, 0(sp)
```

În sens invers, pentru a scoate elementul din vârful stivei și a îl depozita în registrul `s0`, vom folosi:

```
lw s0, 0(sp)
addi sp, sp, 4
```

4.1 Apelul unei proceduri

Pentru a apela o procedură, putem folosi două instrucțiuni:

```
call procedura
jal procedura
```

Aceasta este de fapt o pseudoinstrucțiune pentru instrucțiunea:

```
jal x1, procedura
```

care, ne reamintim, salvează adresa instrucțiunii următoare în registrul **x1**, adică registrul **ra** (return address). Așa cum îi spune și numele acest registru este utilizat pentru reținerea adresei de retur. Așadar spre deosebire de x86, revenirea dintr-o funcție se face bazat pe un registru, valoarea nemaifiind transmisă prin stivă. Returul se face prin instrucțiunea:

```
ret
```

RISC-V mai are o pseudoinstrucțiune de salt pe care nu am prezentat-o și anume:

```
jr reg
```

care produce saltul la adresa reținută în registrul **reg** și este la rândul ei o pseudoinstrucțiune pentru:

```
jalr x0, reg, 0
```

primul operand marcând la fel ca în cazul instrucțiunii **jal** locația în care se va stoca adresa instrucțiunii următoare, iar ultimul operand reprezentând offsetul.

Cum este destul de ușor de remarcat, instrucțiunea de retur este deci doar un alias pentru:

```
jr ra
```

Așadar ca schelet de bază pentru apelul unei proceduri vom avea următoarea secvență:

```
.data
.text
.global main
main:
    call proc    // jal proc
                 // jal ra, proc

    li a7, 93
    li a0, 0
    ecall

proc:

    ret          // jr ra
                 // jalr x0, ra, 0
```

Atenție! Pare că Ripes începe execuția întotdeauna în prima procedură pe care o găsește, motiv pentru care vom pune mereu funcția **main** prima.

4.2 Argumentele unei proceduri și valoarea de retur

În cazul RISC-V, de asemenea, argumentele sunt transmise și ele prin regiștrii conform convenției pe care am văzut-o și la apelurile de sistem. Regiștrii **a0** - **a7** sunt folosiți pentru transmiterea primelor 8 argumente. Dacă există mai mult de 8 argumente, acestea vor fi depozitate pe stivă ca în cazul x86.

De asemenea, valorile returnate se vor menține în regiștrii **a0**, **a1**, iar în cazul în care sunt mai mult de două, vor fi transmise prin vârful stivei.

Exemplu: Fie acum funcția având declarația `proc(x,y,&sum)` care realizează adunarea a doi întregi `x` și `y` și depozitează suma în `sum` și care de asemenea returnează diferența celor două numere. Până în acest punct, un program minimalist în care se cheamă respectiva funcție cu argumentele 2, 3, respectiv adresa unei variabile din memorie, este prezentat mai jos.

```
.data
    x: .long 3
    y: .long 2
    sum: .long 0
    dif: .long 0

.text
.global main
main:
    lw a0, 0(gp)
    lw a1, 4(gp)
    addi a2, gp, 8
    call proc

    sw a0, 12(gp)

    li a7, 93
    li a0, 0
    ecall

proc:
    add a3, a0, a1
    sw a3, 0(a2)
    sub a0, a0, a1

    ret
```

4.3 Crearea cadrului de apel

La fel ca în cazul arhitecturii x86, avem nevoie să marcăm pe stivă acele locații de unde încep să se pună pe stivă elemente din cadrul curent. Registrul folosit pentru a marca începutul acestui "frame" este `fp` (frame pointer-ul). De asemenea, pentru a nu pierde adresa de retur în cazul unor apeluri imbricate, la intrarea în funcție se salvează și registrul `ra` (ambii sunt de fapt regiștrii callee-saved). `fp` este mutat pentru a pointa către acea locație de pe stivă de după aceste două salvări (Ripes acceptă doar denumirea de `s0`).

Astfel o procedură va avea următorul schelet:

```
proc:
    addi sp, sp, -8
    sw ra, 4(sp)
    sw s0, 0(sp)
    addi s0, sp, 0
```

```
// corpul procedurii
```

```
lw s0, 0(sp)
lw ra, 4(sp)
addi sp, sp, 8
ret
```

În plus, conceptul de regiștrii callee-saved/caller-saved funcționează la fel ca pentru x86. Tipul fiecărui registru este menționat în Tabelul 2.

Exercițiu: Observați valorile depozitate în **s3**, respectiv **t5** la finalul execuției programului. Adăugați instrucțiunile necesare pentru a menține și valoarea registrului callee-saved **t5**.

```
.data
x: .long 3
y: .long 2
sum: .long 0
dif: .long 0

.text
.global main
main:
    li s3, 10
    li t5, 11

    lw a0, 0(gp)
    lw a1, 4(gp)
    addi a2, gp, 8
    addi a3, gp, 12
    call proc

final:
    li a7, 93
    li a0, 0
    ecall

proc:
    addi sp, sp, -8
    sw ra, 4(sp)
    sw s0, 0(sp)
    addi s0, sp, 0
    addi sp, sp, -4
    sw s3, -4(s0)

    add s3, a0, a1
    sw s3, 0(a2)

    sub t5, a0, a1
```

```
sw t5, 0(a3)

lw s3, -4(s0)
addi sp, sp, 4

lw s0, 0(sp)
lw ra, 4(sp)
addi sp, sp, 8
ret
```

Exercițiu: Este posibil să facem alocarea de spațiu pe stivă o singură dată, numai că offsetii pentru salvarea lui **ra** și **fp** se modifică. Adaptați programul astfel încât să alocați de la început 12 octeți, iar la final să îi ștergeți.

Mai multe informații despre convențiile de apel puteți găsi aici ¹⁵.

¹⁵<https://d3s.mff.cuni.cz/files/teaching/nswi200/202324/doc/riscv-abi.pdf>

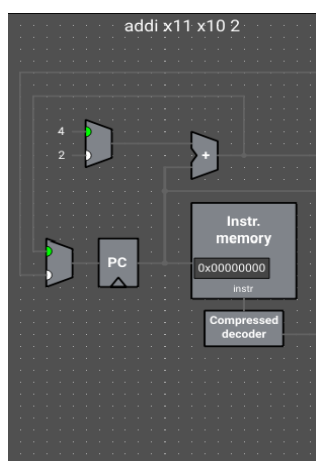
5 Pipelining

Pipeline-ul unui procesor este o tehnică utilizată pentru a spori performanța procesorului, permițând execuția simultană a mai multor instrucțiuni în etape diferite. .

Pipeline-ul este o componentă esențială în arhitectura procesoarelor moderne, fiind prezent în procesoarele RISC (Reduced Instruction Set Computing) și CISC (Complex Instruction Set Computing). Până acum am folosit Ripes având în spate o simulare a unui proces într-un singur ciclu, în procesor putând exista la un anumit moment doar o singură instrucțiune. În continuare vom analiza un procesor în 5 "stage"-uri (primul dintre ele **5-stage processor w/o forwarding or hazard detection**) care este compus din următoarele unități:

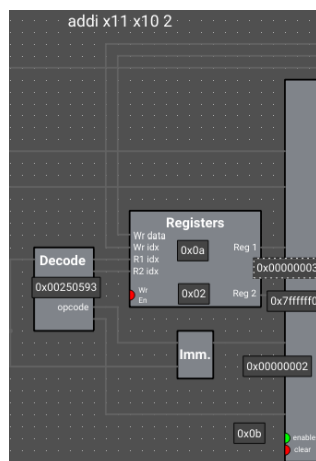
- **Fetch (F)**: Se preia instrucțiunea din memorie.
- **Decode (D)**: Se decodează instrucțiunea pentru a determina ce operație trebuie executată.
- **Execute (E)**: Se execută operația pe baza instrucțiunii folosind unitatea aritmetico-logică.
- **Memory (M)**: Citirea sau scrierea datelor în memoria principală, dacă instrucțiunea implică astfel de operații.
- **Write Back (WB)**: Salvarea rezultatelor în registru sau memorie, pregătindu-le pentru instrucțiunile următoare.

Să considerăm acum un program având o singură instrucțiune `addi a1, a0, 3`, iar `a0 = 2`. Să analizăm în continuare fiecare stage.

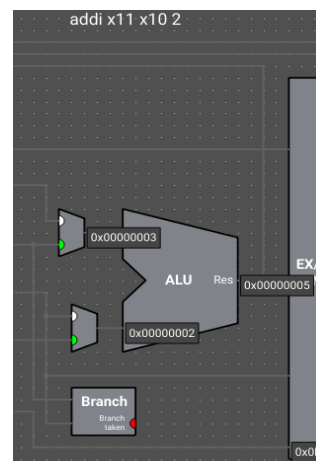


1. În etapa de **fetch**, prin PC (care este 0), este preluată instrucțiunea regăsită la acea adresă în memorie.

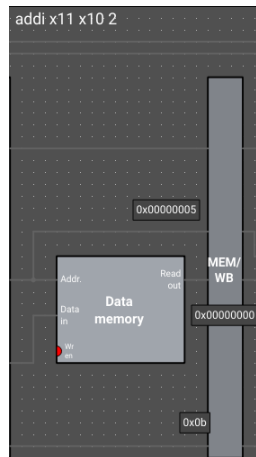
Instrucțiunile comprimate sunt transformate în corespondentele lor necomprimate. Tot aici PC este incrementat cu 2 sau 4 pe baza acestei informații.



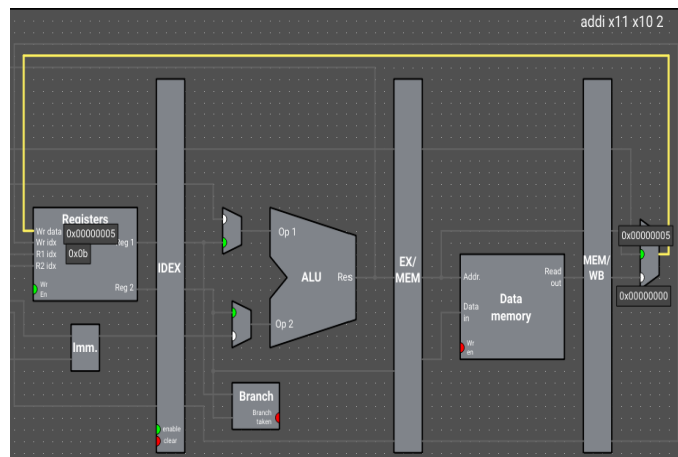
2. În etapa de **decode**, este interpretată encodarea și sunt preluate sursele și destinația operației. Sunt transmise mai departe valorile curente ale celor 2 regiștri sursă (sau 0 dacă nu există un al doilea), respectiv valoarea imediată și encodarea pentru identificarea ulterioară a operației.



3. În etapa de **execute**, cele două valori sunt transmise către ALU unde se aplică operația indicată de către opcode (adunarea cu imediat). Aceasta este o schemă simplificată, pentru a observa și ruta pe care se transmite tipul operației, puteți selecta o variantă de schemă extinsă.



4. În etapa de **memory**, se accesează datele din memorie (dacă ele există). După cum se poate observa, instrucțiunea aceasta nu folosește nimic din memorie, deci se trece prin în această etapă doar transmitând mai departe rezultatul.



5. În etapa de **write back** se produce scrierea efectivă în destinație - fie în memorie (în cazul nostru este 0, nu se accesează memoria), fie în register file, modificându-se valoarea registrului destinație.

În ciuda avantajelor, performanța unui pipeline depinde de mai mulți factori care pot introduce penalizări.

5.1 Hazard-uri în pipeline

Un hazard este o problemă care împiedică procesorul să execute eficient instrucțiunile în pipeline. Există trei tipuri principale:

1. **Hazarduri de date** - Apar când o instrucțiune are nevoie de rezultatele unei alte instrucțiuni care nu a fost încă finalizată. Ca potențiale soluții remarcăm:
 - (a) **Data forwarding:** Rezultatele din etapele intermediare sunt "transmise înainte" pentru a fi utilizate fără a aștepta finalizarea completă.
 - (b) **Stalling (blocare):** Pipeline-ul este temporar oprit pentru a aștepta disponibilitatea datelor.
 - (c) **Out-of-order execution:** Un procesor OoO decuplează strict ordinea de preluare și decodificare a instrucțiunilor (care rămân in-order) de ordinea în care acestea sunt executate (care este out-of-order). Prin reorganizarea execuției instrucțiunilor, procesorul poate reduce latențele introduse de hazarduri. La final, instrucțiunile sunt menținute într-un buffer, iar rezultatele lor sunt finalizate și scrise în registre în ordinea originală (in-order commit).
2. **Hazard-uri de control** - Apar în cazul instrucțiunilor de salt (branching), când procesorul nu știe ce instrucțiune să preia până când condiția saltului este evaluată. Pentru a elimina din penalități putem folosi **Branch prediction** - prezicerea direcția saltului și începerea execuției

instrucțiunilor prezise. Dacă predicția este greșită, instrucțiunile incorecte din pipeline sunt eliminate.

3. **Hazard-uri structurale** - Apar când două sau mai multe instrucțiuni încearcă să folosească aceeași resursă hardware în același timp (ex. o singură unitate de acces la memorie). Pentru a evita acest aspect, este nevoie ca proiectarea procesorului să se realizeze cu resurse hardware redundante.

Exercițiu: Considerați următoarea declarație în memorie `.long 200, 300, 400`. Analizați pe rând pipeline-ul pentru instrucțiunile `lw a0, 4(gp)`, `sw a0, 8(gp)` respectiv `beq a0, a1, 12`.

Să încercăm în continuare să observăm pipeline-ul când avem un cod cu mai multe instrucțiuni:

```
.data
.long 100, 200, 300
.text
.global main
main:
    lw a0, 4(gp)
    lw a1, 0(gp)
    add a0, a0, a1
    sw a0, 8(gp)
```

Exercițiu: Care este rezultatul depus în memorie? Ce s-a întâmplat?

Să observăm de asemenea codul următor:

```
.data
.long 100, 200, 300
.text
.global main
main:
    lw a0, 0(gp)
    lw a1, 4(gp)
    blt a0, a1, label

    sw a0, 8(gp)
    j final

label:
    sw a1, 8(gp)

final:
    li a7, 93
    li a0, 0
    ecall
```

Exercițiu: Care este rezultatul depus în memorie? Ce s-a întâmplat?

Pentru rezolvarea acestor probleme, sunt oferite două variante îmbunătățite de procesor care abordează una dintre următoarele metode:

1. **Pipeline stall** (5-Stage processor w/o forwarding unit) - sunt introduse **nop**-uri (stall-uri) în conducta de date până când sursele sunt updatate.
2. **Forwarding/Bypassing** (5-stage processor w/o hazard detection) - sunt transmise datele din ALU direct către următoarea instrucțiune înainte de a fi scrise.

În cazul acesta însă cea de-a doua variantă nu este în continuare de folos, fiind necesar totuși mai mulți cicli între accesări. Pentru eficiență și corectitudine, vom folosi deci procesorul care implementează ambele mecanisme (5-stage processor).

Exercițiu: Observați avantajul mecanismului de forwarding (5-Stage processor w/o forwarding unit vs 5-stage processor).

În plus, Ripes oferă și o schemă logică extinsă pentru un procesor superscalar - dual issue care permite intrarea simultană în pipeline a două instrucțiuni. În plus față de procesorul anterior, acesta are 6 stage-uri, adăugându-se etapa **Issue** în care instrucțiunile sunt distribuite către unitățile funcționale disponibile (ALU, unități de acces la memorie).

5.2 Măsurarea performanței

1. **Throughput** - Măsoară câte instrucțiuni pot fi finalizate pe unitatea de timp. (5-stage processor - o instrucțiune, 6-stage dual-issue processor - 2 instrucțiuni).
2. **Latența** - Timpul total necesar pentru a finaliza o singură instrucțiune (din Fetch până la Write Back - 5 vs 6).
3. **CPI (Cycles Per Instruction)** - numărul mediu de cicluri necesare pentru a finaliza o instrucțiune. Într-un procesor pipelined, CPI ideal = 1, dar în realitate poate fi mai mare din cauza hazard-urilor.
4. **IPC (Instructions Per Cycle)** - numărul efectiv de instrucțiuni executate pe ciclu de ceas. Un procesor superscalar (dual-issue sau quad-issue) poate avea un IPC mai mare de 1 ($IPC = 1/CPI$).
5. **Utilizarea pipeline-ului** - procentul de timp în care pipeline-ul este complet ocupat cu sarcini utile. Idle time (timp de inactivitate) înseamnă că resursele nu sunt utilizate eficient.

Exemplu: Un program execută un număr egal de instrucțiuni aritmetice, de accesare a memoriei și cu floating point. Știm că o instrucțiune aritmetică ia 4 cicli, una de accesare a memoriei 5, iar una care accesează unitatea de floating point 6. Cât este **CPI**? Dar **IPC**? **Observație:** Acest tip de calcul ignoră multe alte variabile.

Răspuns:

$$CPI = 0.33 * 4 + 0.33 * 5 + 0.33 * 6 = 4.95$$

$$IPC = 1/4.95 = 0.2020$$

6 Cache

Memoria cache este un tip de memorie rapidă utilizată pentru a crește performanța procesorului, reducând timpul necesar pentru a accesa datele frecvent utilizate din memoria principală (RAM). Cache-ul este amplasat de obicei în apropierea procesorului și joacă un rol crucial într-un sistem de calcul modern.

Cache-ul funcționează ca o memorie intermediară între procesor și memoria principală, stocând datele utilizate recent sau predicții de date ce vor fi necesare în viitor. Prin aceasta, el reduce latența accesului la memorie și sporește eficiența execuției instrucțiunilor.

Cache-ul exploatează principiile **localității spațiale** (datele din apropierea unei locații utilizate recent sunt probabil necesare în viitor) și **localității temporale** (datele accesate recent sunt probabil utilizate din nou în scurt timp).

Exercițiu: Considerăm un sistem de calcul pe 32 de biți care are un cache de 16 Kbytes. Considerăm că memoriile sunt împărțite în blocuri de 32 de bytes. Câte blocuri sunt posibile în memoria principală? Câte blocuri sunt în memoria cache?

Memoria cache este organizată în mai multe niveluri ierarhice, fiecare având caracteristici distincte (L0, L1, L2 etc.). Dispozitivele cu capacități reduse pot să nu aibă deloc sau să aibă mai puține niveluri de memorie cache. Simulatorul din Ripes de exemplu are doar memorie cache L1 pe care o vom folosi pentru a descoperi principalele caracteristici ale unei astfel de memorii.

O să începem prin a selecta o memorie cu 32 de linii sau blocuri și cu o dimensiune a liniei de 4 word-uri (16 bytes), direct-mapped (prima variantă preset). Să urmărim mai întâi algoritmul pentru descoperirea corespondenței dintre o adresă din memoria principală și locația din cache.

Să considerăm codul de mai jos (recomand folosirea variantei de procesor **5-stage processor** care pare a fi cea mai stabilă pentru prezentarea noțiunilor de cache):

```
.data
.long 0x10, 0x11, 0x12, 0x13
.long 0x14, 0x15, 0x16, 0x17
.long 0x18, 0x19, 0x20, 0x21
.text
.global main
main:
    lw a0, 20(gp)      #0x10000014 --> 0x15
    lw a1, 24(gp)      #0x10000018 --> 0x16
    add a0, a0, a1
    sw a0, 28(gp)      #0x1000001c --> 0x17
    lw a1, 0(gp)       #0x10000010 --> 0x10
    add a0, a0, a1
    sw a0, 32(gp)      #0x10000020 --> 0x18

final:
    li a7, 93
    li a0, 0
```

ecall

Exercițiu: Vom considera $gp = 0x10000000$, valoarea inițială indicată de Ripes. Observați poziționarea în memoria cache. Puteți deduce regula?

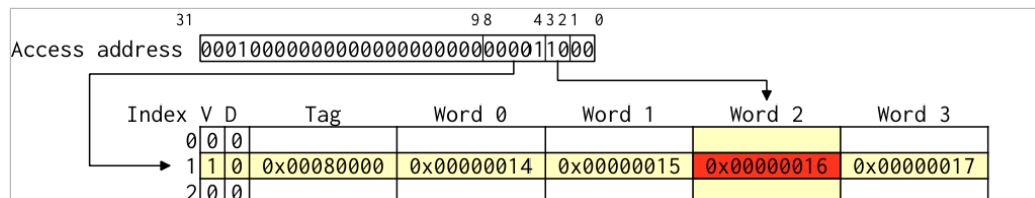


Figura 4: Linie de cache din Ripes

Algoritm

1. Unitatea de bază este word-ul, știm așadar ca ultimii 2 biți vor fi mereu 0 (adresele sunt din 4 în 4).
2. Extragerea offset-ului - Se determină câți biți sunt rezervați pentru offset pe baza dimensiunii blocului. În cazul nostru, o linie conține 16 bytes, 4 word-uri, deci offsetii vor lua valori de la 0 la 3 - avem nevoie de încă 2 biți.
3. Extragerea indexului - Indexul indica linia/blocul din cache, deci va depinde de numărul de linii/blocuri din cache. În acest caz, avem 32 de linii, deci avem nevoie de 5 biți pentru identificare. Acesta este folosit pentru a identifica locația din cache în care s-ar putea afla adresa cerută din memorie.
4. Extragerea tag-ului - Tag-ul este reprezentat de restul de biți rămași. Tag-ul este comparat cu tag-ul stocat în acea locație din cache pentru a verifica dacă datele sunt cele căutate.

Exercițiu: Urmăriți cum se modifică memoria cache pe parcursul execuției programului. Care ar fi size-ul minim al unei linii pentru a avea cel mai bun hit rate?

Exercițiu: Care este corelația dintre dimensiunea cache-ului ca număr de linii, și dimensiunea unei linii pe de-o parte și conceptele de localitate temporală și spațială pe de cealaltă parte?

În plus față de aceste concepte, putem modifica și **tehnica de mapare**:

- **Mapare directă** - Fiecare bloc din memoria principală poate fi mapat doar într-o locație specifică din cache. Simplu de implementat, dar poate duce la conflicte frecvente.
- **Mapare asociativă** - Orice bloc din memoria principală poate fi mapat în orice locație din cache. Este mai flexibil, dar implică un cost mai mare pentru căutare.
- **Mapare x-way set asociativă** - Combinație între maparea directă și cea asociativă, unde cache-ul este împărțit în seturi, iar un bloc din memoria principală poate fi mapat într-un subset specific de locații din cache.

Exercițiu: Schimbați tipul de mapare în mapare asociativă. Ce observați?

Să considerăm acum următorul program:

```
.data
.long 0x10, 0x11, 0x12, 0x13
.long 0x14, 0x15, 0x16, 0x17
.long 0x18, 0x19, 0x20, 0x21
.text
.global main
main:
    lw a0, 8(gp)
    lw a1, 12(gp)
    add a2, a0, a1
    lw a0, 24(gp)
    lw a1, 28(gp)
    add a3, a0, a1
    lw a0, 40(gp)
    lw a1, 44(gp)
    add a4, a0, a1
final:
    li a7, 93
    li a0, 0
    ecall
```

Exercițiu: Să considerăm acum o memorie cache 2-way set asociative cu un line size de 2 și cu doar 2 linii de cache. Ce observați că se întâmplă?

Când cache-ul este plin, procesorul trebuie să decidă de asemenea ce bloc să înlocuiască. Politicile obișnuite includ:

- **LRU (Least Recently Used)** - Se înlocuiește blocul care nu a fost folosit de cel mai mult timp.
- **FIFO (First In First Out)** - Se înlocuiește blocul care a fost adăugat primul.
- **Random** - Blocul înlocuit este ales aleatoriu.

O altă caracteristică a cache-ului (din păcate nefuncțională pe Ripes) este politica de scriere:

1. **Write-through** - Procesorul modifică datele în blocul corespunzător din cache, iar modificarea este simultan scrisă și în memoria principală.
2. **Write-back** - Procesorul scrie datele doar în blocul din cache și setează un indicator numit bitul dirty (dirty bit). Dacă blocul trebuie înlocuit din cache, datele sunt scrise în memoria principală dacă acest bit este setat.

O altă politică adițională pentru scriere este **(No) Write Allocate** - decizia de a aduce blocul în memoria cache și la scriere sau nu.

6.1 Performanța memoriei cache

Performanța cache-ului este determinată de:

1. **Rata de accesare (hit rate)** - Procentajul accesărilor care găsesc datele necesare în cache. Rata ridicată de accesare reduce semnificativ latența memoriei.
2. **Latența memoriei** - Timpul necesar pentru a accesa datele. Latența este minimă pentru L1 și crește pentru L2, L3 și memoria principală.
3. **Penalty-ul ratării (miss penalty)** - Timpul suplimentar necesar pentru a obține datele din memoria principală atunci când acestea nu sunt prezente în cache.
4. **Politici de preluare (prefetching)** - Procesorul poate prelua proactiv date care ar putea fi necesare în viitor, bazându-se pe modele de acces..

7 Exerciții

7.1 Laboratorul 9

1. Se dau două numere naturale x și y . Să se scrie un program care să realizeze interschimbarea lor.
2. Se dau 4 numere a, b, c și d . Să se calculeze rezultatul operației $((a + b + 5) * 2) \ll c | d$ și să se afișeze pe ecran.
3. Să se indice encodarea pentru instrucțiunea `and t0, s1, a5`.

7.2 Laboratorul 10

1. Afișați numărul de divizori ai unui întreg stocat în memorie și apoi o listă cu aceștia. ("Numărul 6 are 4 divizori: 1, 2, 3, 6").
2. Realizați o procedură `divizori(int x)` care realizează afișarea de mai sus și întoarce numărul de divizori.
3. Realizați o procedură `divizori_elemente(int *v, int n, int k)` care parcurge vectorul v cu n elemente și returnează numărul de elemente ce au exact k divizori. Afișați în `main` un mesaj de tipul "Sunt nr elemente cu exact k divizori". Folosiți-vă de procedura din exercițiul anterior.

7.3 Laboratorul 11

1. Un procesor are un pipeline cu 5 stadii: Fetch, Decode, Execute, Memory, Write-back. Determinați câte cicluri sunt necesare pentru a executa 4 instrucțiuni dacă nu apar hazarduri. Ce impact ar avea un hazard de date între instrucțiunile 1 și 2 asupra ciclurilor totale de execuție?
2. Să considerăm un procesor cu următoarea frecvență a instrucțiunilor și următoarele costuri:
 - ALU pentru întregi - 50%, 1 ciclu
 - Load - 20%, 5 cicluri
 - Store - 10%, 1 ciclu
 - Branch - 20%, 2 cicluri

Calculați CPI și decideți care schimbare ar aduce cea mai bună performanță dintre următoarele:

- implementarea unui mecanism de branch prediction pentru a reduce costul branch-urilor la 1 ciclu;
- folosirea unui data cache care are reduce costul unui load la 3 cicluri.

3. Dacă un procesor superscalar este introdus pentru a rezolva hazardurile de date, câte cicluri pot fi economisite pentru următoarea secvență de instrucțiuni:

```
ADD R1, R2, R3
SUB R4, R1, R5
MUL R6, R7, R8
```

4. Memoria principală are o dimensiune de 2^{16} bytes, iar cache-ul are o capacitate de 1 KB, cu o dimensiune a blocului de 32 bytes. Calculați numărul total de blocuri din memoria principală. Calculați numărul de linii de cache. Determinați unde va fi mapată adresa 0x3A7F în cache, conisderând cache-ul direct-mapped.
5. Un cache asociativ cu 4 seturi și 2 linii per set cu size-ul unei linii 1 utilizează politica LRU (Least Recently Used). Considerați următoarele adrese accesate: 0x0001, 0x0005, 0x0021, 0x0005, 0x0045. Explicați cum sunt organizate datele în cache după fiecare acces. Care bloc este înlocuit atunci când adresa 0x0045 este accesată?

8 Resurse suplimentare

- <https://riscv.org/educational-resources/>
- <https://www.allaboutcircuits.com/technical-articles/introductions-to-risc-v-instruction-set-understanding-this-open-instruction-set-architecture/>
- <https://www.cse.iitd.ac.in/~srsarangi/archbook/chapters/riscv.pdf>
- https://passlab.github.io/CSE564/notes/lecture09_RISCV_Impl_pipeline.pdf
- <https://nsec.sjtu.edu.cn/data/MK.Computer.Organization.and.Design.4th.Edition.Oct.2011.pdf>